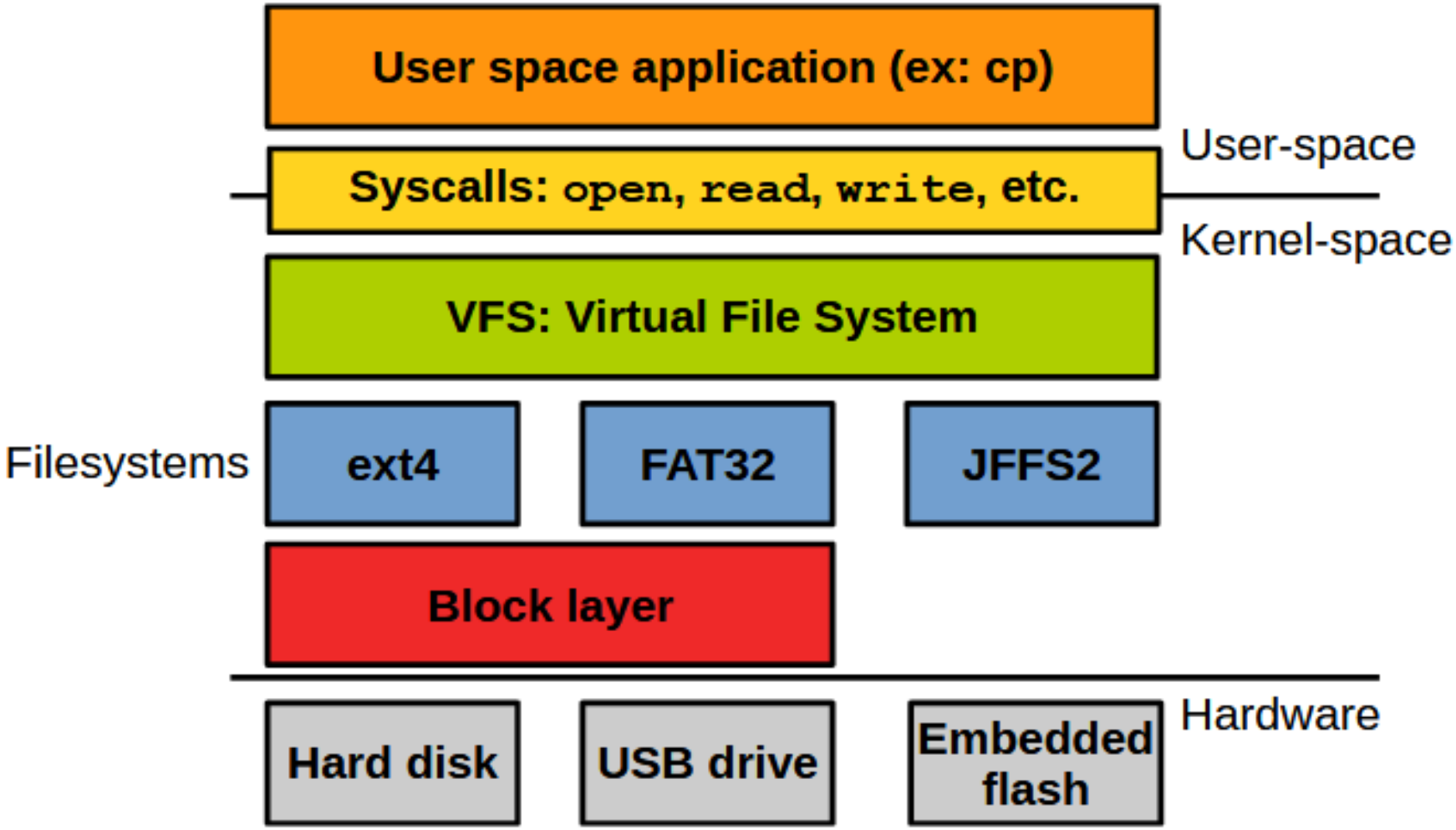
The block I/O layer
I/O schedulers

Table of contents

Block layer



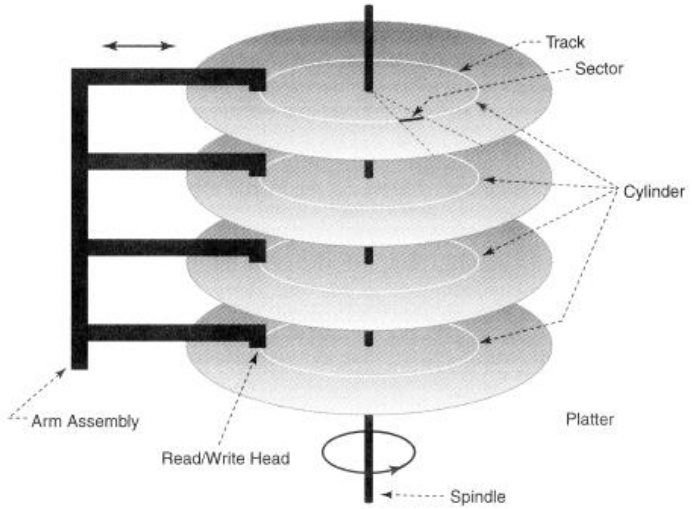
Source: https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/l/lec22-block-layer

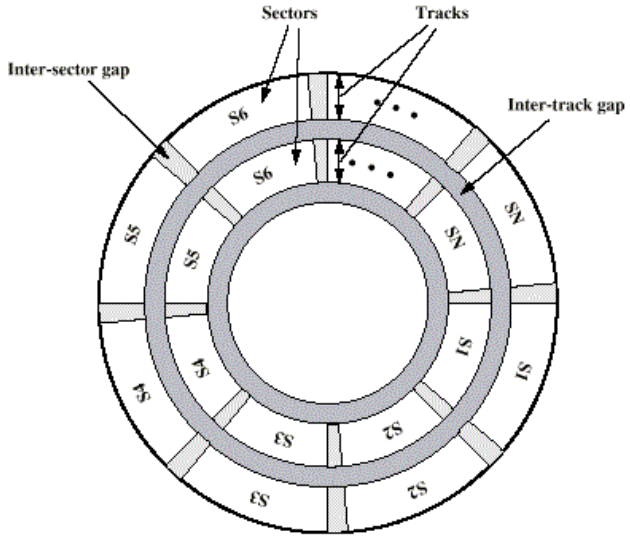
Anatomy of a block device

- Block devices: HDD, SSD, CD/DVD, Fibre Channel (FC) SAN, or other block-based storage device.
- Disk structure:
  - **Sector**: minimum addressable unit in a block device.
  - **Track**: a set of all sectors on a single surface lying at the same distance from the disk's rotation spindle.
  - **Cylinder**: a set of all tracks lying at the same distance from the disk's rotation spindle on a disk with multiple platters.

request handling time =

seek time + latency time + transmission time





Source: Internet

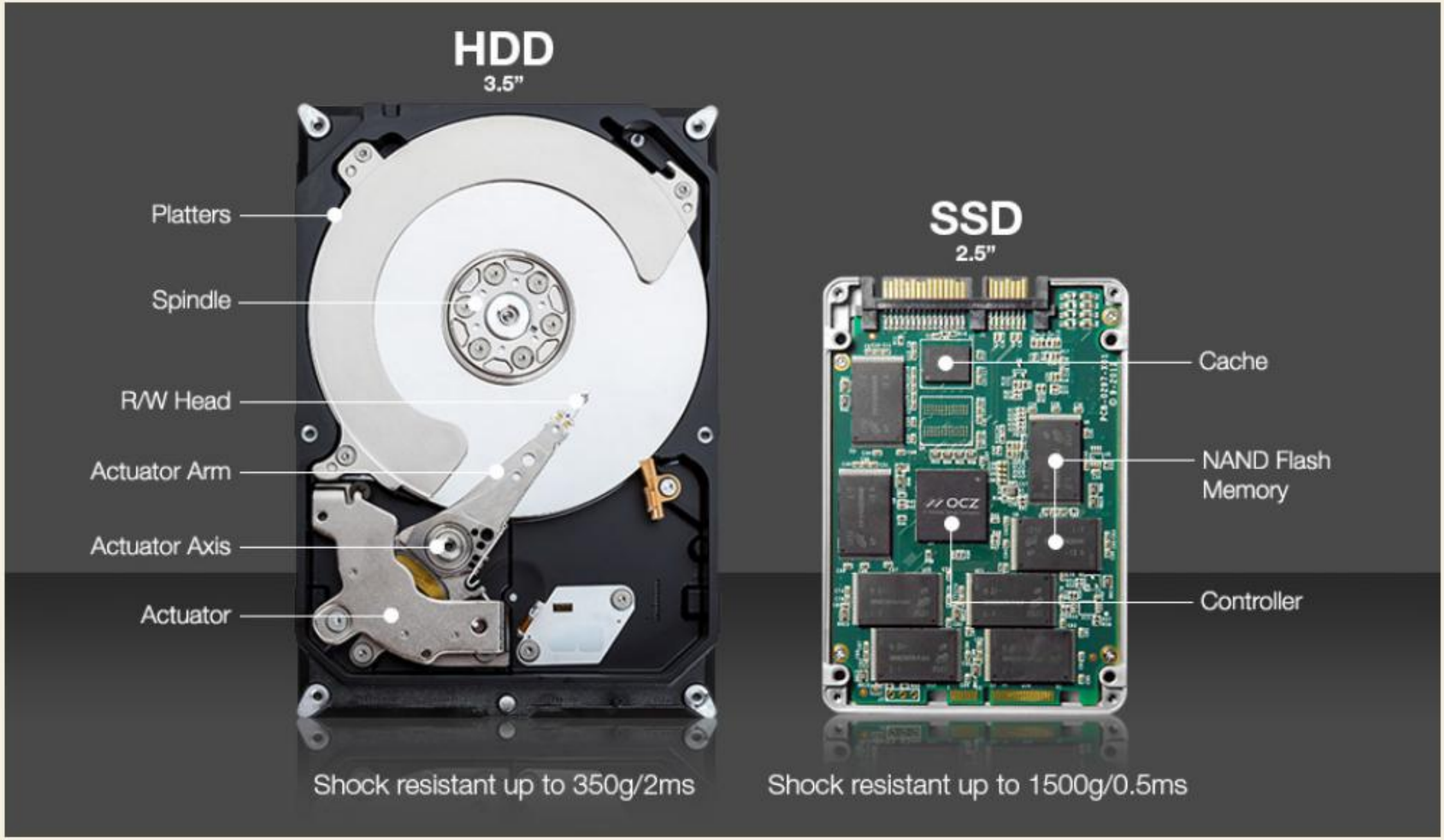


Photo: partitionwizard.com

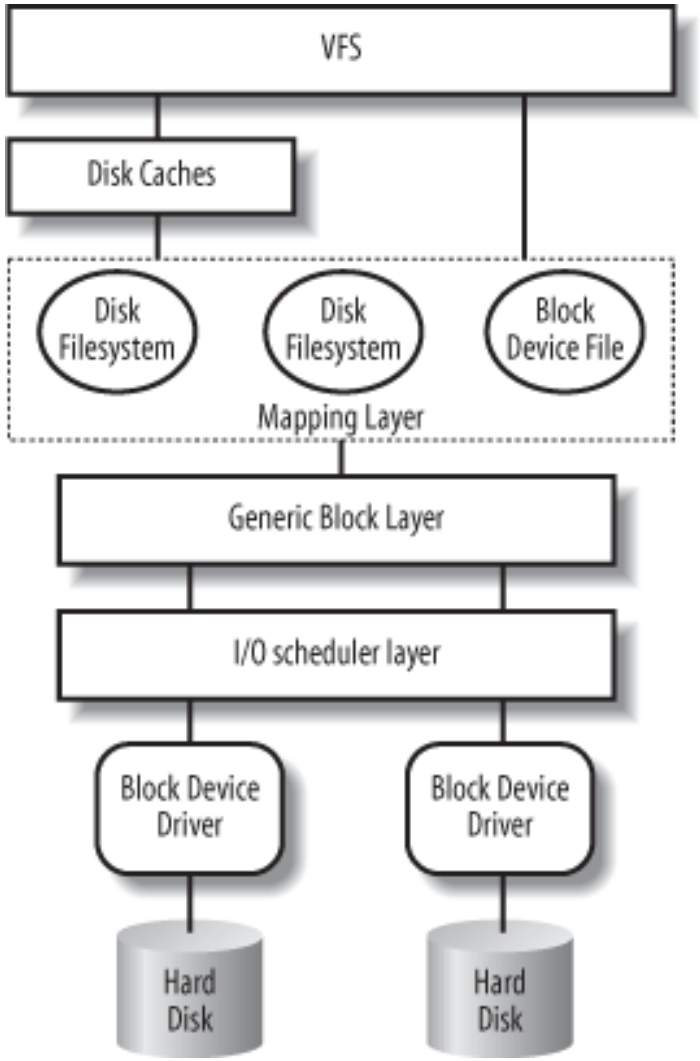
Handling block devices

1. The **read ()** system function transfers control to the appropriate function from **VFS**. The VFS function determines if the requested data is already **available** and, if necessary, how to **perform** the **read** operation.

2. If the kernel must **read** the data from the **block device**, it must determine the **physical location** of that data. The kernel relies on the **mapping layer**, which:

   – determines the **block size** of the **filesystem** including the file and computes the extent of the requested data in terms of **file block numbers**.

   – invokes a filesystem-specific function that accesses the **file's disk node** and determines the position of the requested data on disk in terms of **logical block numbers**.



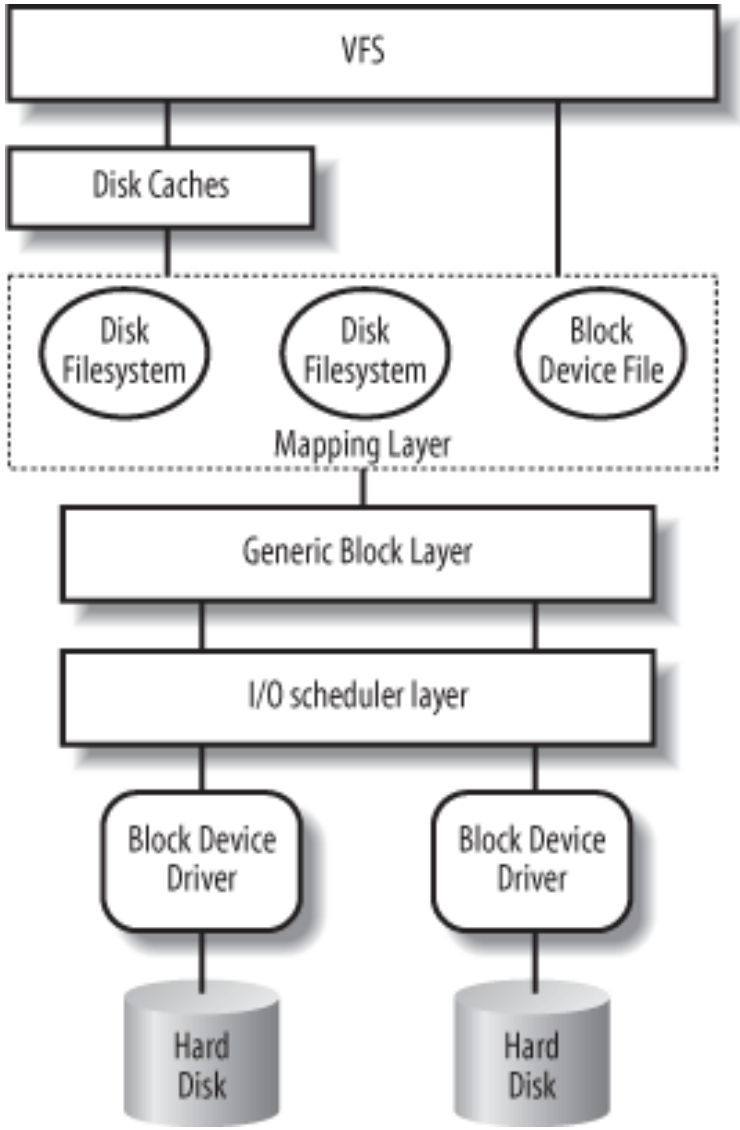
Kernel components affected by a block device operation (source: Bovet, Cesati)

Handling block devices

4. **Generic block layer** starts the I/O operations. Each **I/O operation** involves a group of blocks that are **adjacent on disk**. Because the requested data is not necessarily adjacent on disk, the generic block layer might start **several I/O operations**. Each I/O operation is represented by a **block I/O (bio) structure**, which collects all information needed by the lower components to satisfy the request.

5. The **I/O scheduler** sorts the pending I/O data transfer requests according to predefined kernel policies. The purpose is to group requests of data that lie near each other on the physical medium.

6. The **block device drivers** take care of the actual data transfer by sending suitable commands to the hardware interfaces of the **disk controllers**.



Kernel components affected by a block device operation (source: Bovet, Cesati)

Handling block devices

1. The controllers of the **hardware block devices** transfer data in chunks of fixed length called **sectors**.

   The I/O scheduler and the block device drivers must manage **sectors of data**. In most disk devices, the **size of a sector** is **512** bytes.

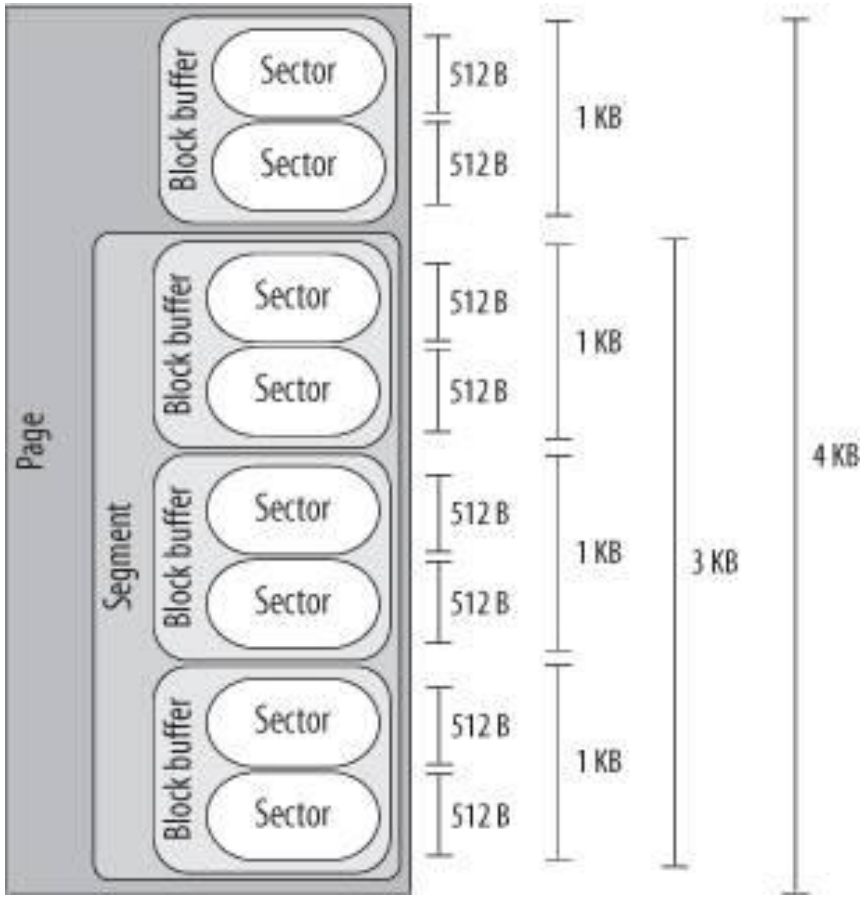
2. **VFS**, the **mapping layer**, and the **filesystems** group the disk data in logical units called **blocks**.

   The block on disk corresponds to one or more **adjacent sectors**, which are regarded by the VFS as a **single data unit**.

   The **block size** cannot be larger than a page frame.

   The block size is **not specific** to a **block device**.

   Each block requires its own **block buffer** in RAM.



Typical layout of page including disk data (source: Bovet, Cesati)

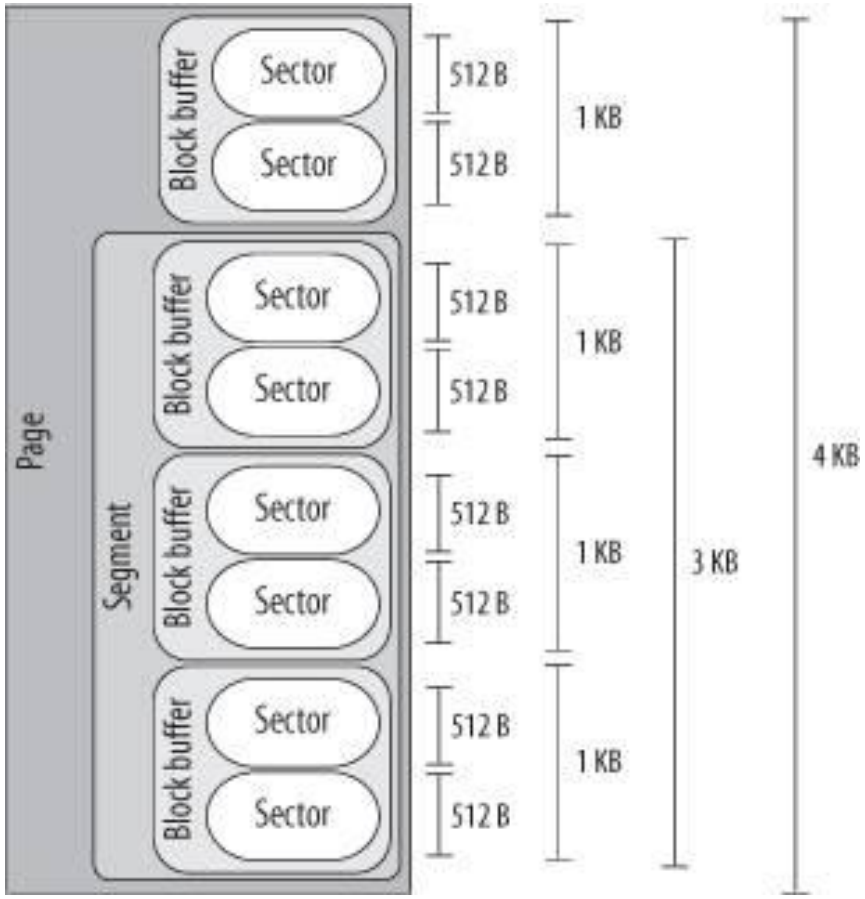
Handling block devices

3. Older disk controllers support simple DMA operations only: data is transferred from /to memory cells that are **physically contiguous in RAM**.

   Recent disk controllers may also support **scatter-gather DMA transfers**. Such DMA operation may involve several segments at once.

   A **segment** is a **memory page** — or a **portion of a memory page**—that includes the data of some **adjacent disk sectors**.

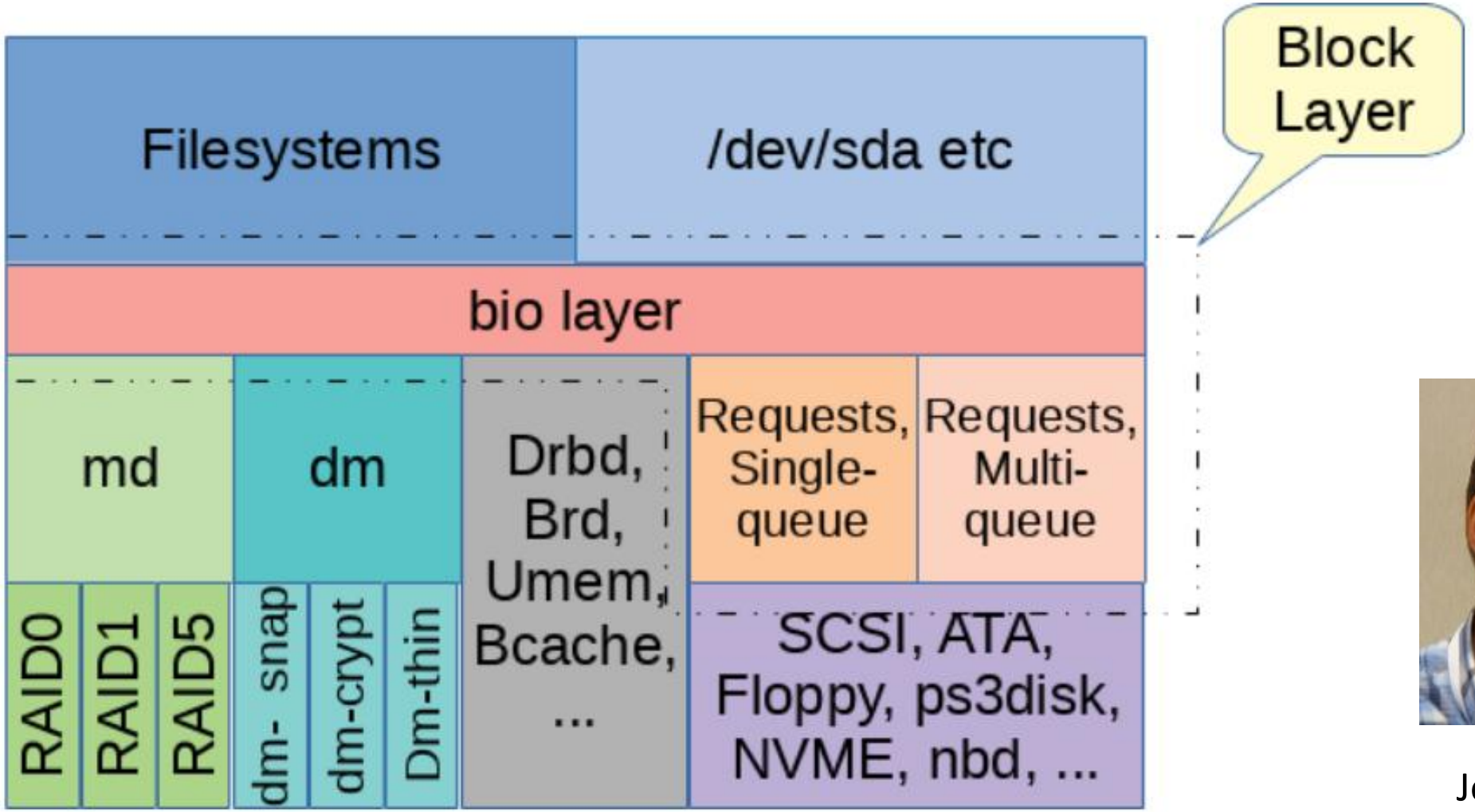
4. The disk caches work on **pages** of disk data, each of which fits in a **page frame**.



The **generic block layer** has to know about **sectors**, **blocks**, **segments**, and **pages** of data.

Typical layout of page including disk data (source: Bovet, Cesati)

A closer look at the block layer



Jens Axboe

Source: https://lwn.net/Articles/736534/

Representing disks and disk partitions

A **disk** is a **logical block device** that is handled by the **generic block layer**. Usually corresponds to **a hardware block device**, but can also be a **virtual device** built upon **several physical disk partitions**, or a **storage area** living in some dedicated **pages of RAM**.

A disk is represented by the **gendisk** object (a generic disk).

A **gendisk** can be associated with multiple **block_device** structures when it has a **partition table**.

There will be one **block_device** that represents the **whole gendisk**, and possibly some others that represent **partitions** within the **gendisk**.

```
struct gendisk {
    int major;                  /* major number of driver */
    int first_minor;
    int minors;                 /* maximum number of minors, =1 for */
                                /* disks that can't be partitioned. */

    char disk_name[DISK_NAME_LEN];   /* name of major driver */

    /* Array of pointers to partitions indexed by partno.  */
    struct xarray part_tbl;

    const struct block_device_operations *fops;
    struct request_queue *queue;
   ...
}
```

Simplified version

I/O scheduling on block devices

I/O requests to **block devices** are handled **asynchronously** at the **kernel level**, and **block device drivers** are **interrupt-driven**.

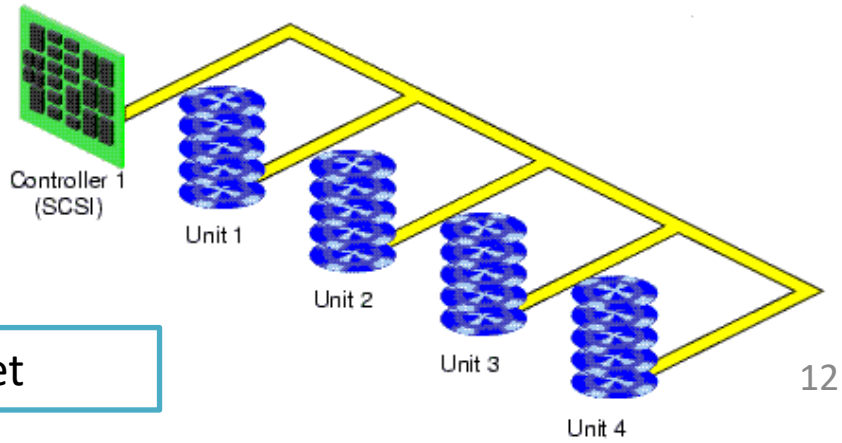
The **generic block layer** invokes the **I/O scheduler** to create a new **block device request** or to enlarge an already existing one and then **terminates**.

The **block device driver** invokes the **strategy routine** to select a pending request and satisfy it by issuing suitable commands to the **disk controller**.

When the I/O operation terminates, the **disk controller** raises an **interrupt** and the handler invokes the strategy routine again to process another pending request.

Each **block device driver** maintains its own **request queue**, which contains the list of pending requests for the device.

If the disk controller is handling **several disks**, there is usually **one request** queue for each **physical block device**. I/O scheduling is performed separately on each request queue.



Source: Internet

I/O scheduling on block devices

The **request queue** is **doubly linked list** of **request descriptors** (struct **request**).

The ordering of elements in the queue is **specific** to each block device driver; the I/O scheduler offers several **predefined** scheduling methods.

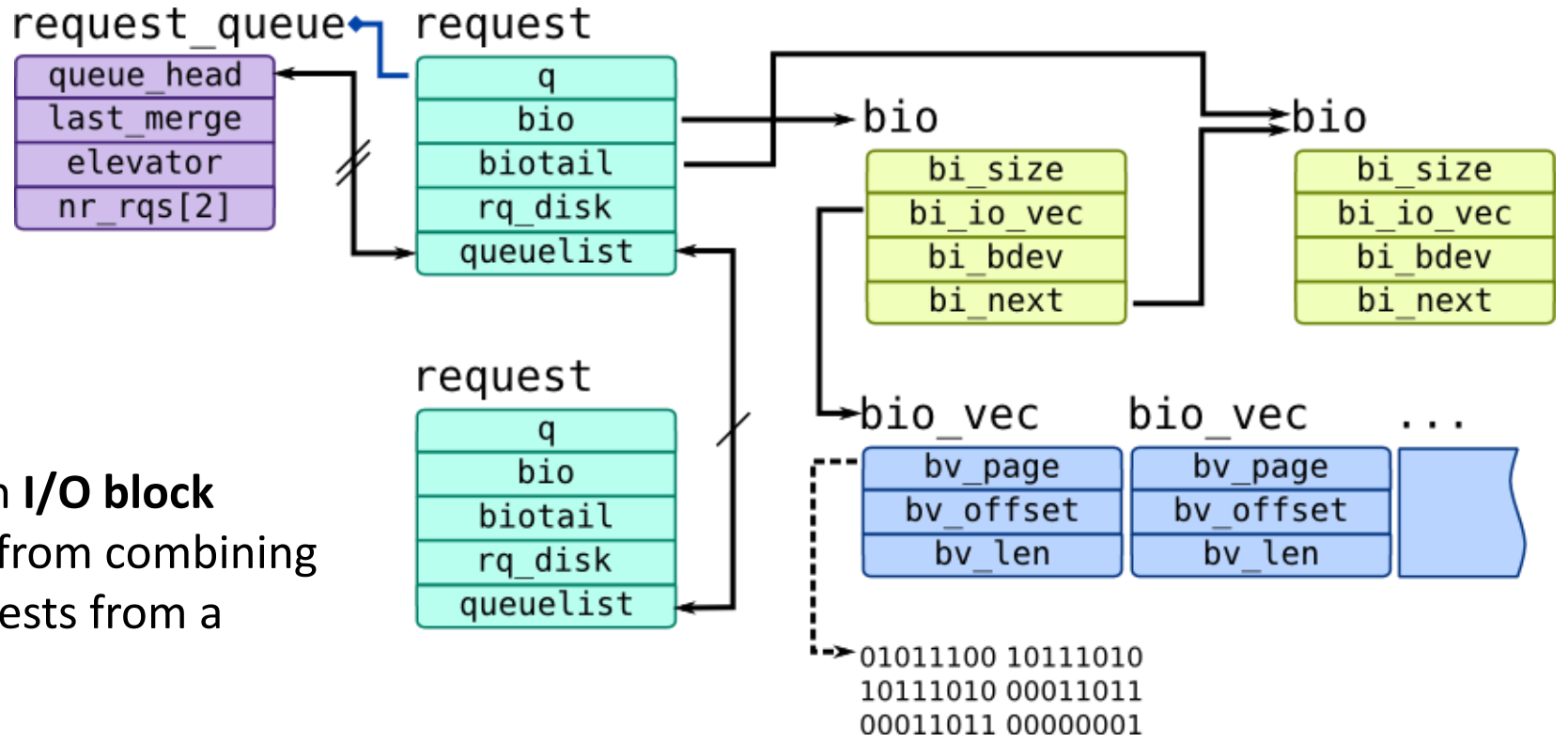
Each **request** consists of one or more **bio structures**.

- Initially, the generic block layer creates a request including just one bio.
- Later, the I/O scheduler may "extend" the request either by **adding a new segment to the original bio**, or by **linking another bio structure into the request**. This is possible when the new data is **physically adjacent** to the data already in the request.

Each **request queue** has a maximum number of allowed pending requests. If the number of pending read (write) requests reaches the upper limit, the **queue** is marked as **full**, and **blockable processes** trying to add requests for that data transfer direction are put to **sleep**.

The **I/O scheduler** determines the exact position of the new request in the queue.

Request queues and requests



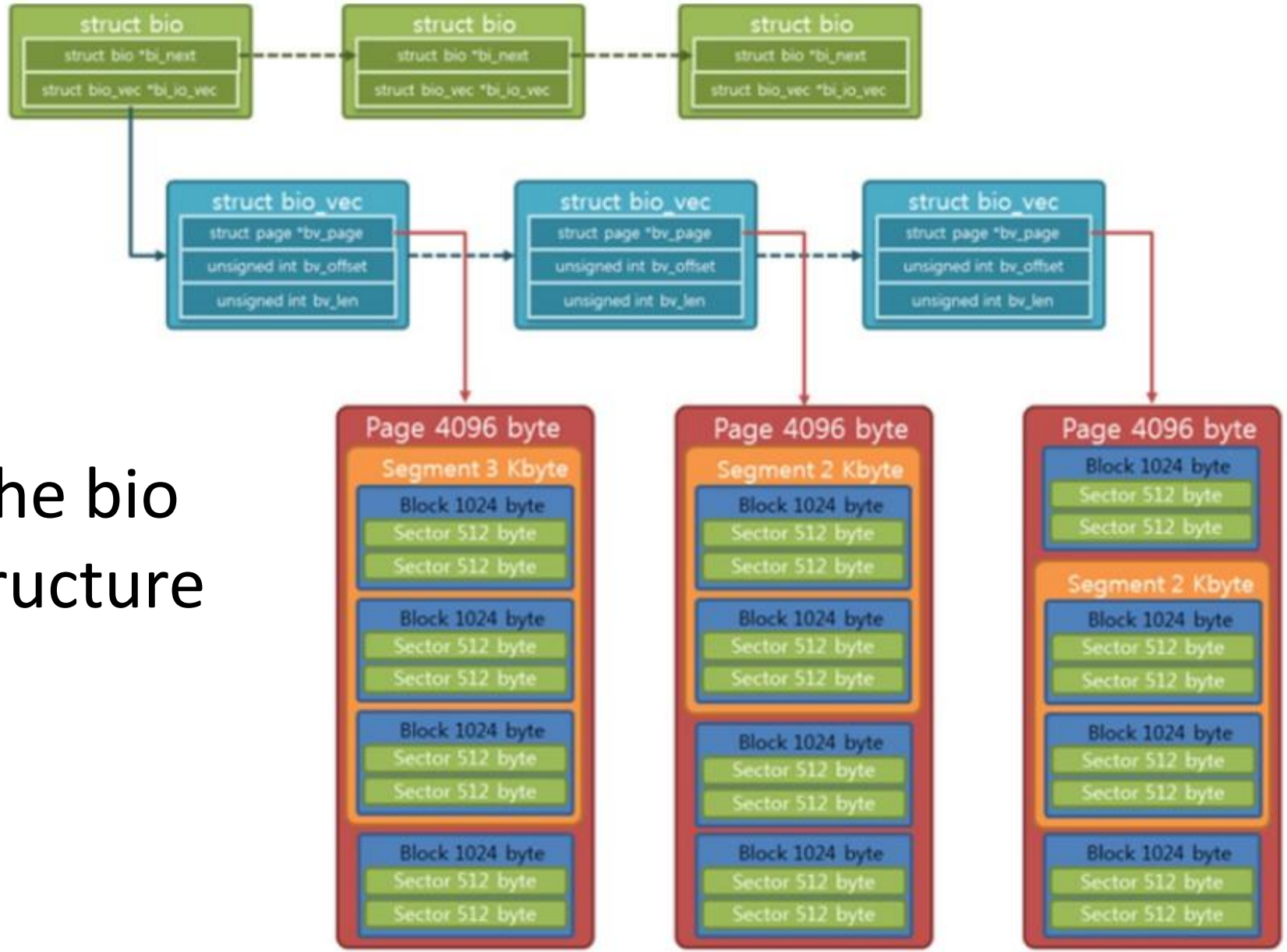
Each **struct request** is an **I/O block request**, but may come from combining more independent requests from a higher level.

The sectors to be transferred for a request can **be scattered into the main memory** but they always correspond to a set of **consecutive sectors on the device**.

Each **bio** includes the initial sector number and the number of sectors included in the storage area, and one or more **segments** describing the memory areas involved in the I/O operation.

VFS objects (source: Sergey Klyaus)

The bio structure



Each **bio vec** represents a page in memory, by default it is **4096 bytes**.

Each **BIO structure** can contain a maximum of 256 bio vec structures.

The bio structure (source: https://hyunyoung2.github.io/2016/09/08/Block_Device/)

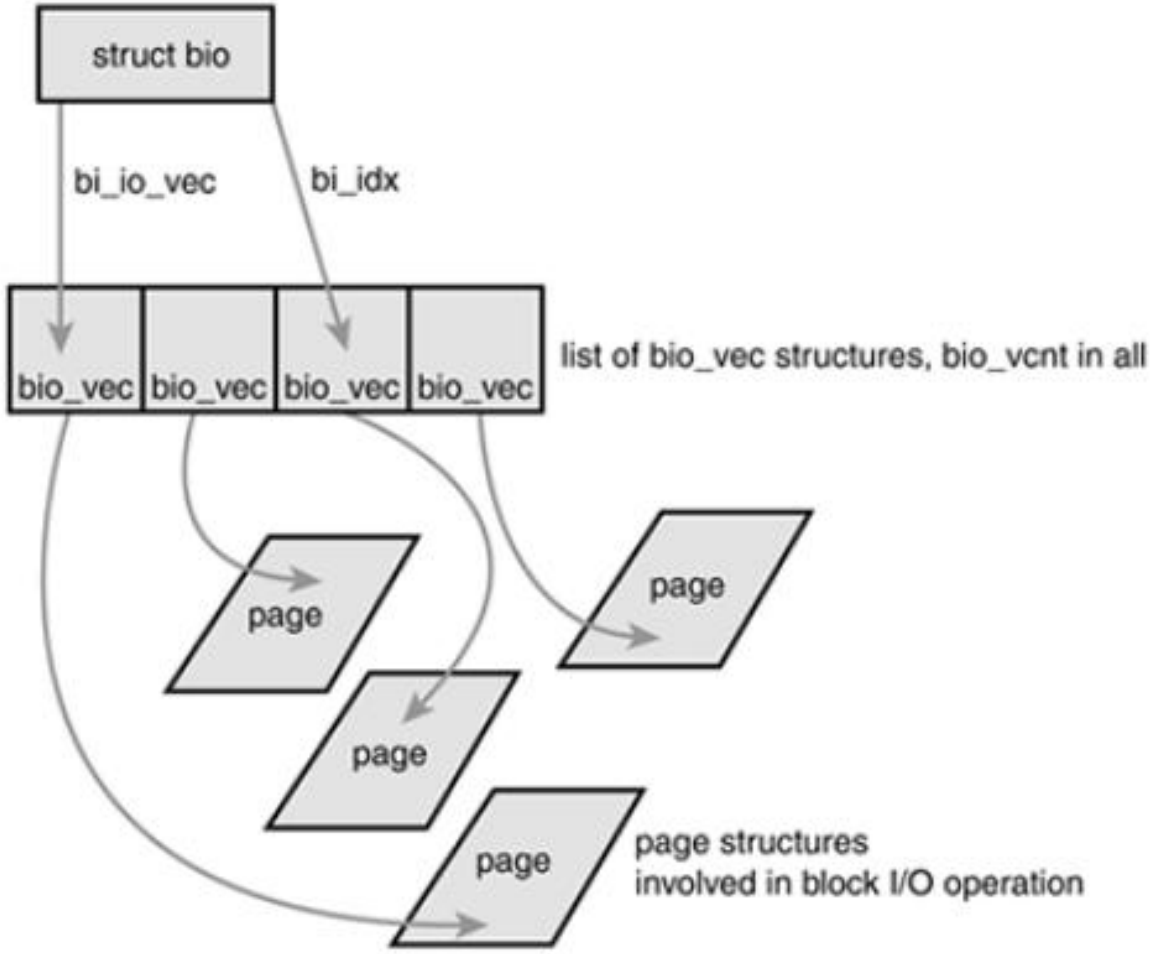
The bio structure

SKIP

The **bio_vec** structure contains a **pointer** to the **page descriptor** of the **segment's page frame**, **length** of the segment in bytes, and **offset** of the segment's data in the page frame.

To iterate through a **struct bio**, we need to iterate through the vector of **struct bio_vec** and transfer the data from every physical page.

To simplify vector iteration, the **struct bvec_iter** is used. This structure maintains information about **how many buffers** and **sectors** were **consumed** during the **iteration**.



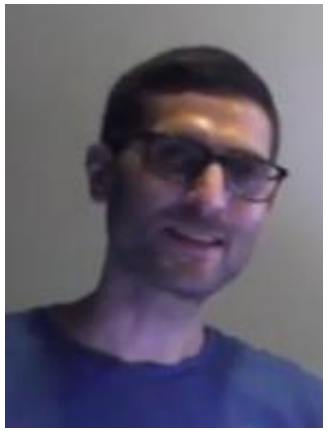
Relationship between struct bio, struct bio_vec, and struct page (source: Bovet, Cesati)

Where were we 10 years ago

- Only rotating storage devices exist.
- Devices were limited to hundreds of IOPs
- Devices access latency was in the milliseconds ballpark
- The Linux block layer was sufficient to handle these devices

- High performance applications found clever ways to avoid storage access as much as possible

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018
Active contributor to Linux I/O and RDMA stack
Interested in video: enjoy (*if you know the language*)



What happened? (hint: HW)

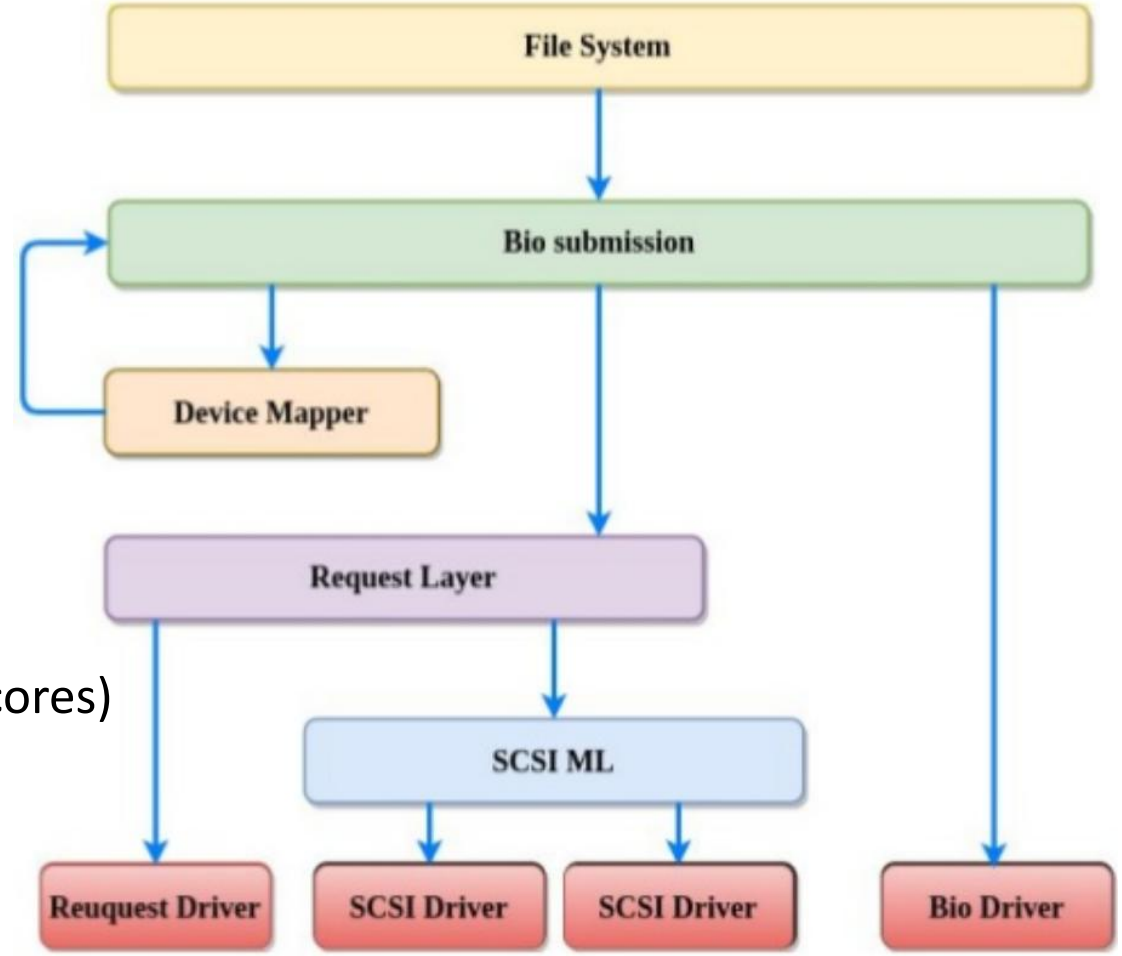
- Flash SSDs started appearing in the DataCenter
  - IOPs went from *Hundreds* to *Hundreds of thousands* to *Millions*
  - Latency went from *Milliseconds* to *Microseconds*
  - Fast Interfaces evolved: PCIe (NVMe)

- Processors core count increased a lot!
  - And NUMA...

NVMe is a high-performance, NUMA optimized, and highly scalable storage protocol, that connects the host to the memory subsystem. The protocol is relatively new, feature-rich, and designed from the ground up for non-volatile memory media (NAND and Persistent Memory) directly connected to CPU via PCIe interface.

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

I/O stack



Existing I/O stack had a lot of data sharing

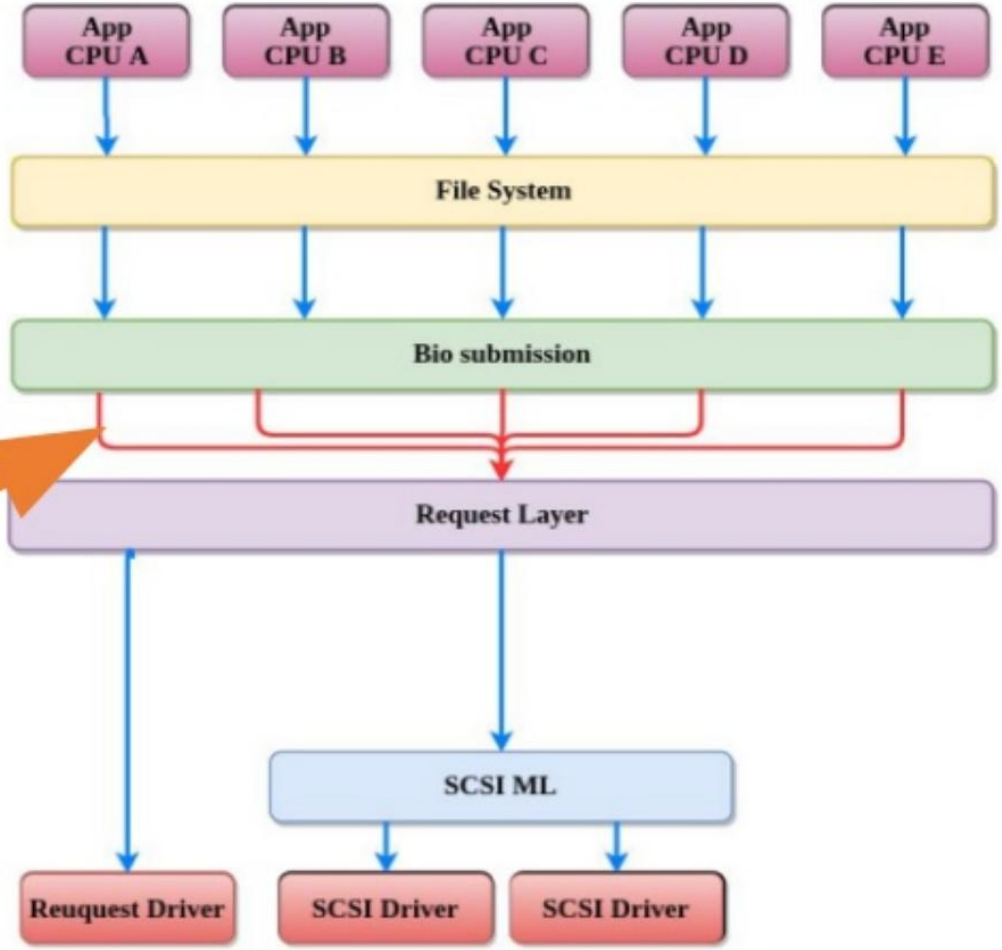
- between different applications (running on different cores)
- between submission and completion
- locking for synchronization
- zero NUMA awarness

All stack heuristics and optimizations centered around slow storage.

The result is very bad scaling, spending lots of CPU cycles and much higher latencies.

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

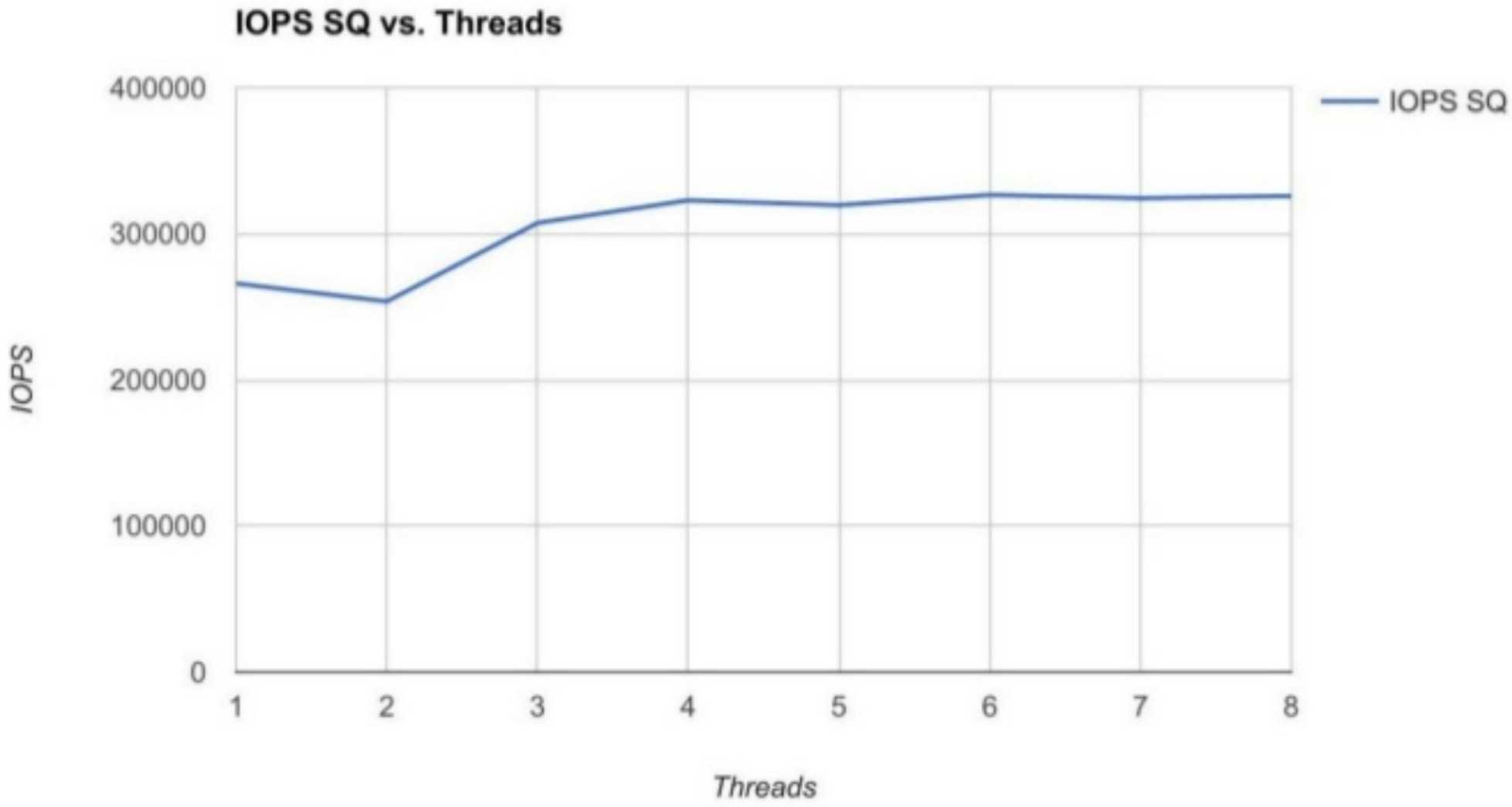
I/O Stack - Little deeper



- Request are serialized
- Placed for staging
- Retrieved by the drivers

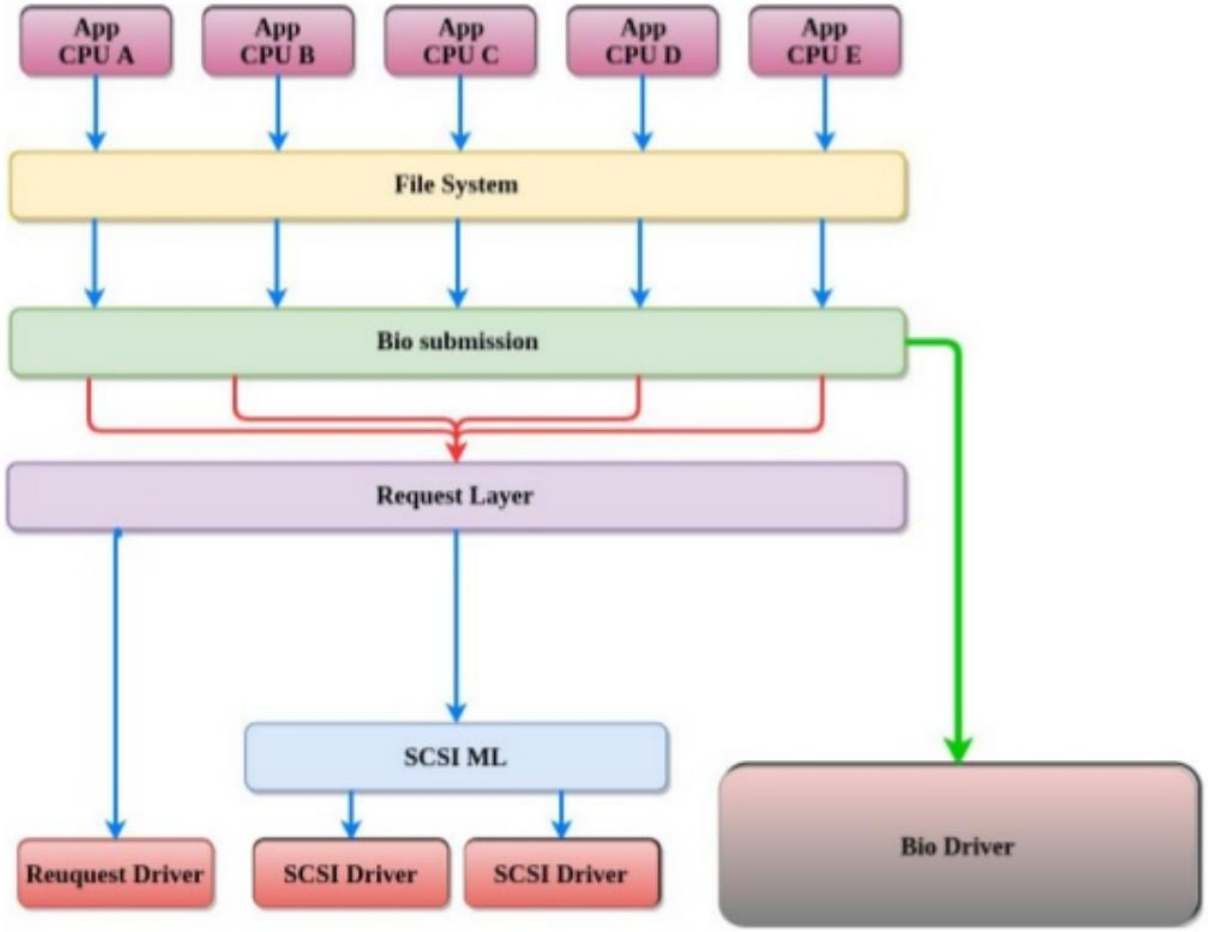
⇒ Lots of shared state!

[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

Workaround: Bypass the the request layer



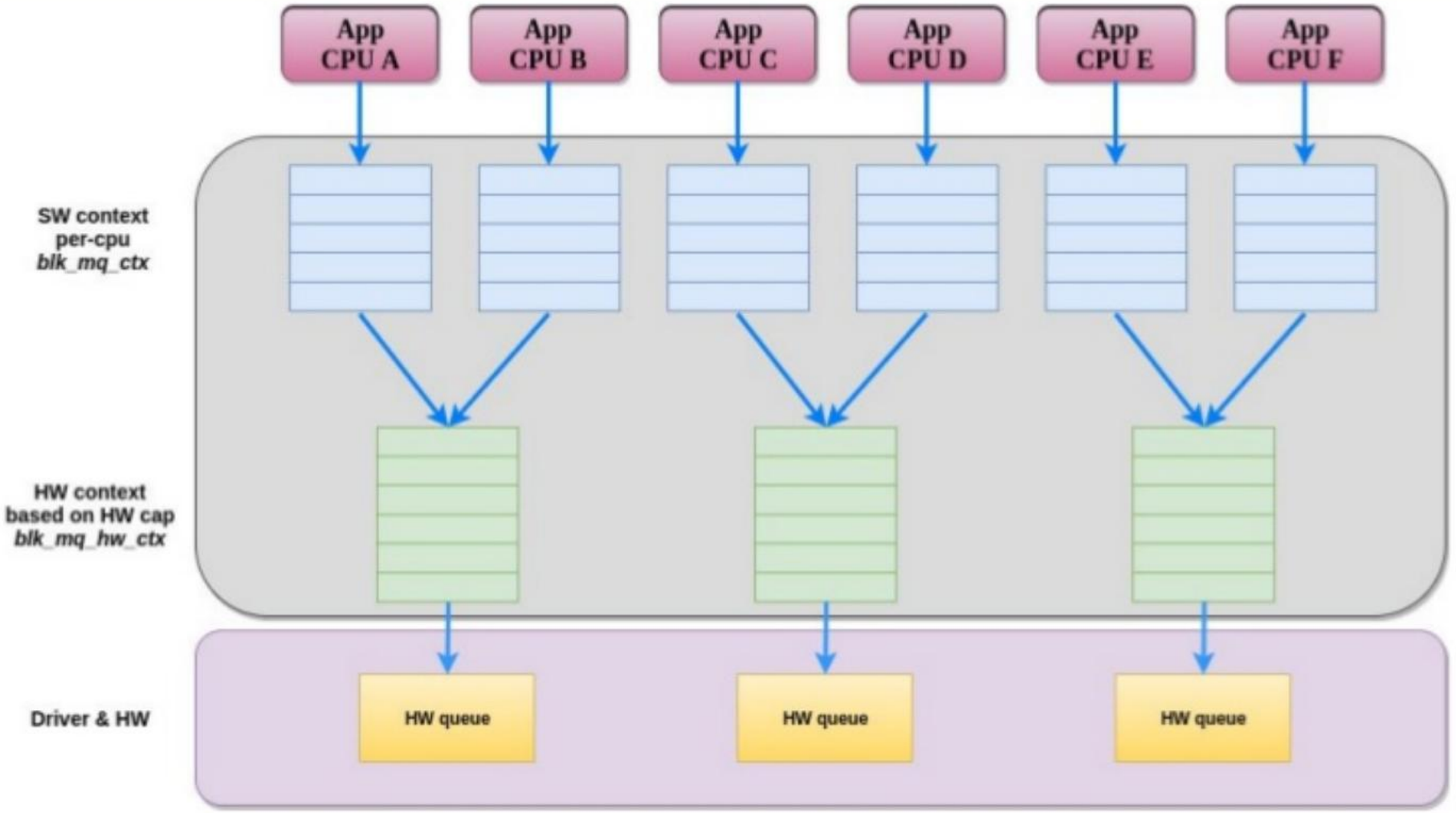
Problems with bypass:

- Give up flow control
- Give up error handling
- Give up statistics
- Give up tagging and indexing
- Give up I/O deadlines
- Give up I/O scheduling
- Crazy code duplication - mistakes are copied because people get stuff wrong...

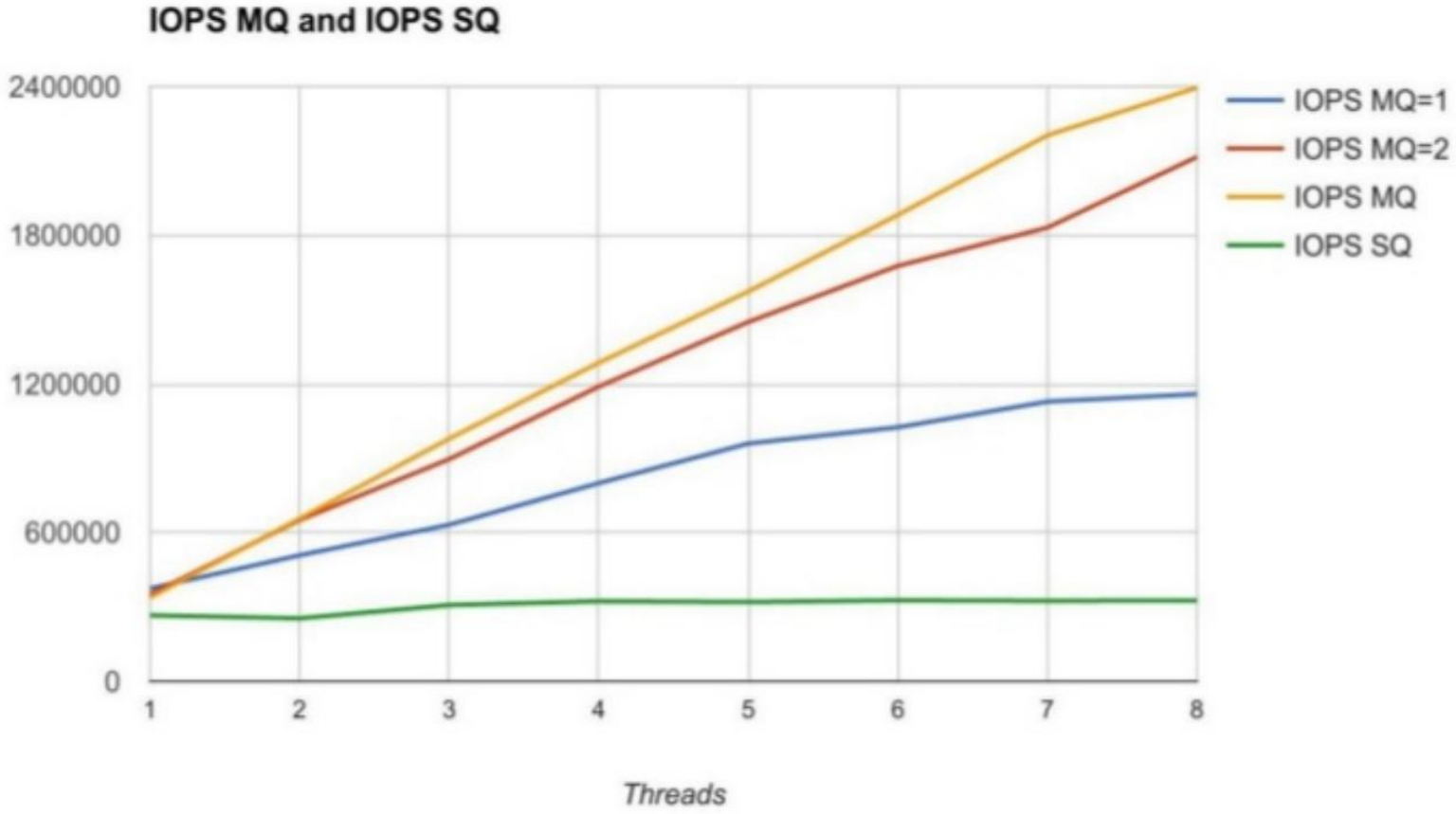
Most importantly, this is not the Linux design approach!

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

Block Multiqueue - Architecture



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018



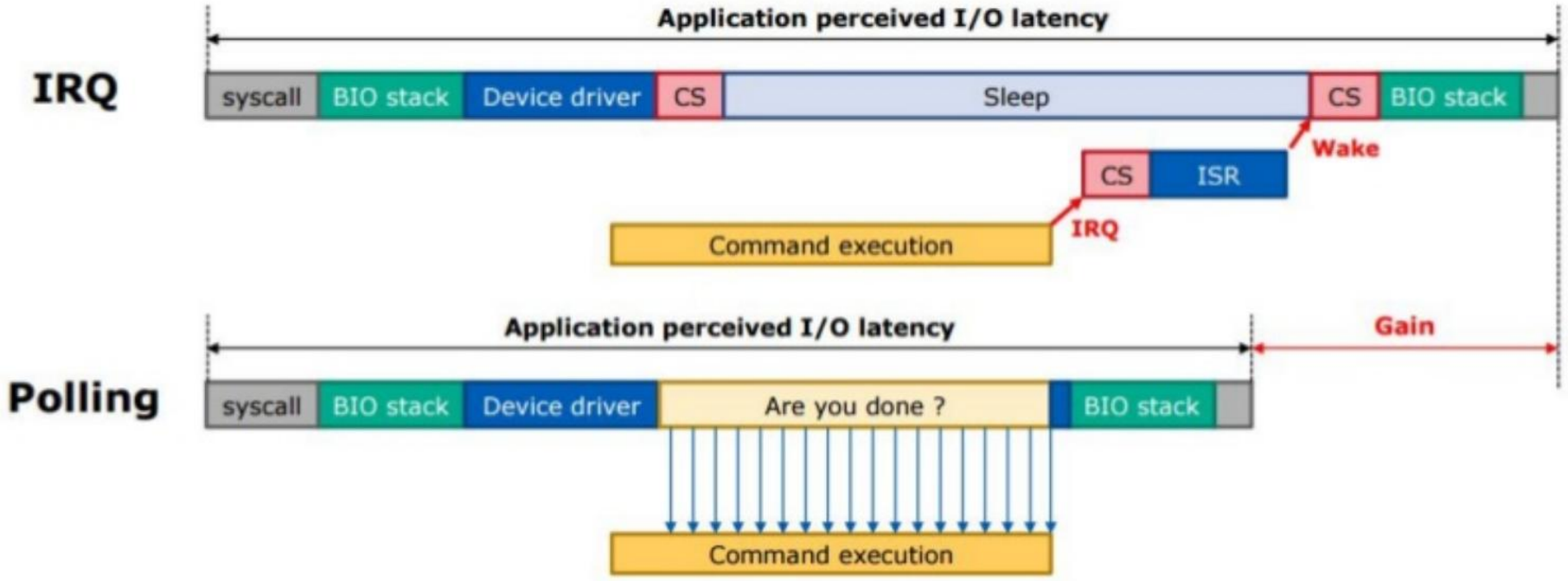
Test-Case:

- null_blk driver
- fio
- 4K sync random read
- Dual socket system

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

IRQ vs. Polling

- Polling can remove the extra context switch from the completion handling



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

But wait, what about I/O schedulers?

- What we didn't mention was that block multiqueue lacked a proper I/O scheduler for approximately 3 years!
- A fundamental part of the I/O stack functionality is scheduling
  - To optimize I/O sequentiality - Elevator algorithm
  - Prevent write vs. read starvation (i.e. deadline scheduler)
  - Fairness enforcement (i.e. CFQ)

- One can argue that I/O scheduling was designed for rotating media
  - Optimized for reducing actuator seek time

NOT NECESSARILY TRUE - Flash can benefit scheduling!

The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

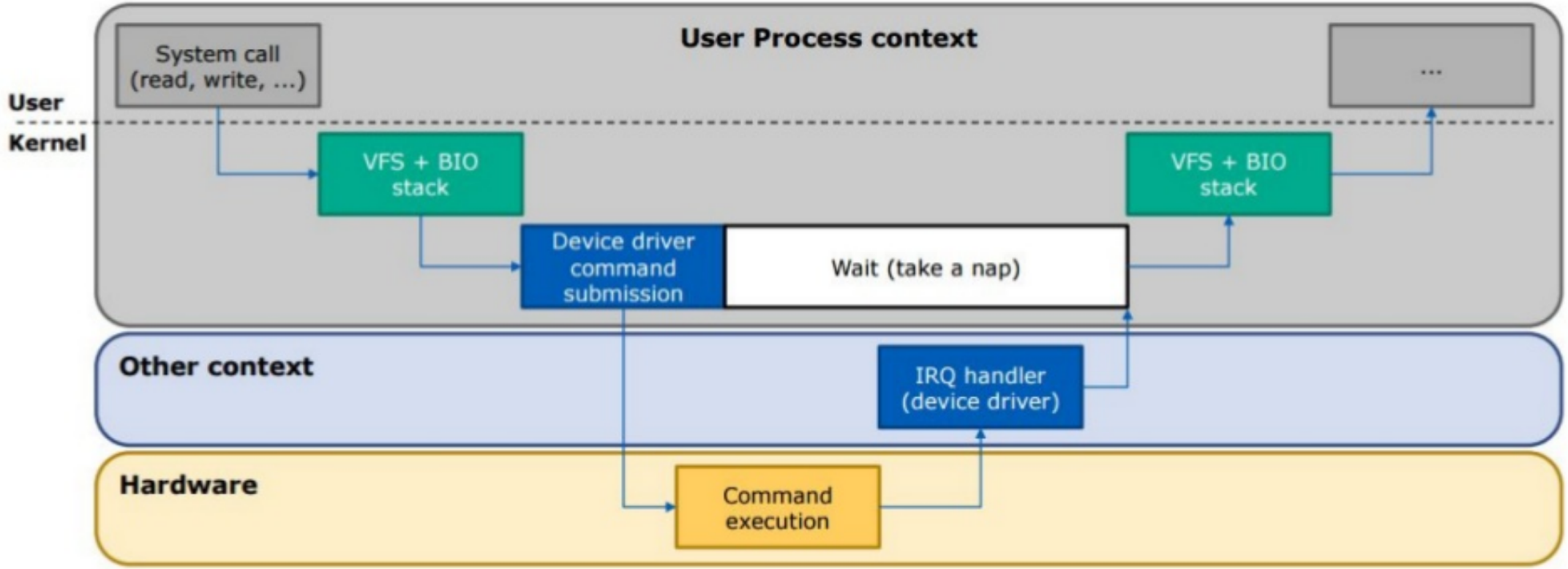
But wait #2: What about Ultra-low latency devices

- New media is emerging with Ultra low latency (1-2 us)
  - 3D-Xpoint
  - Z-NAND

- Even with block MQ, the Linux I/O stack still has issues providing these latencies
  - It starts with IRQ (interrupt handling)
  - If I/O is so fast, we might want to poll for completion and avoid paying the cost of MSI(X) interrupt

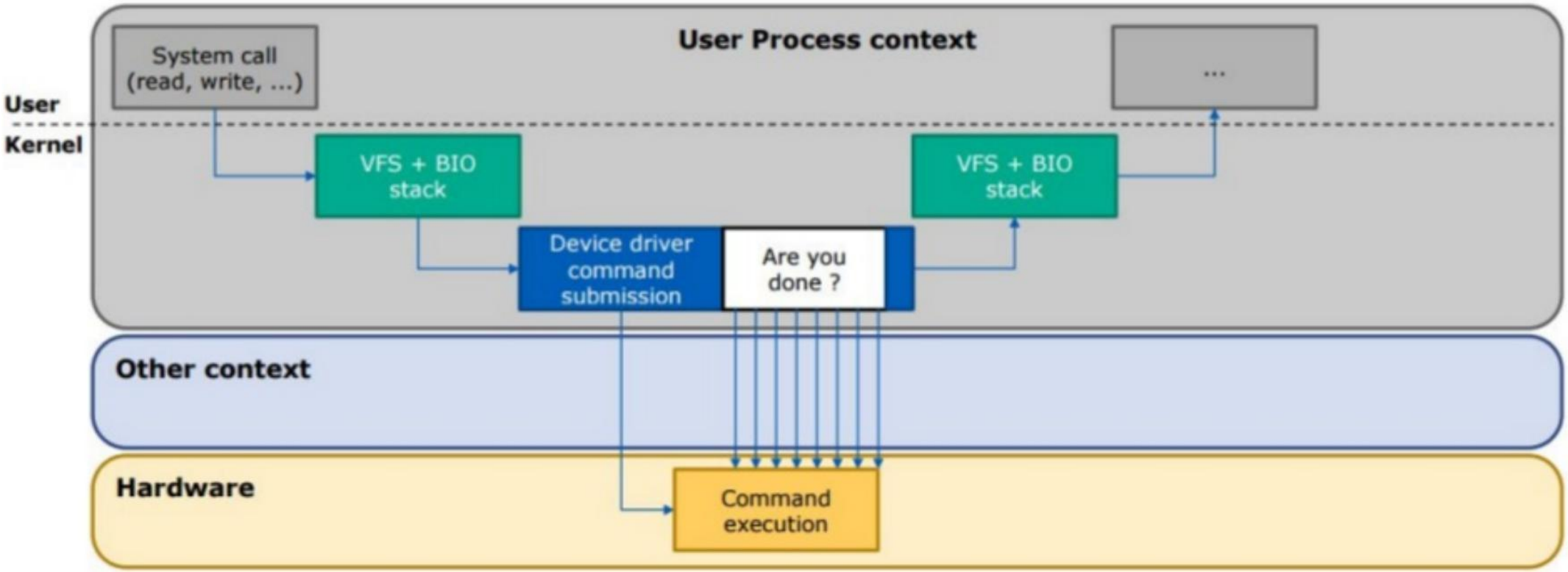
The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

Interrupt based I/O completion model



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

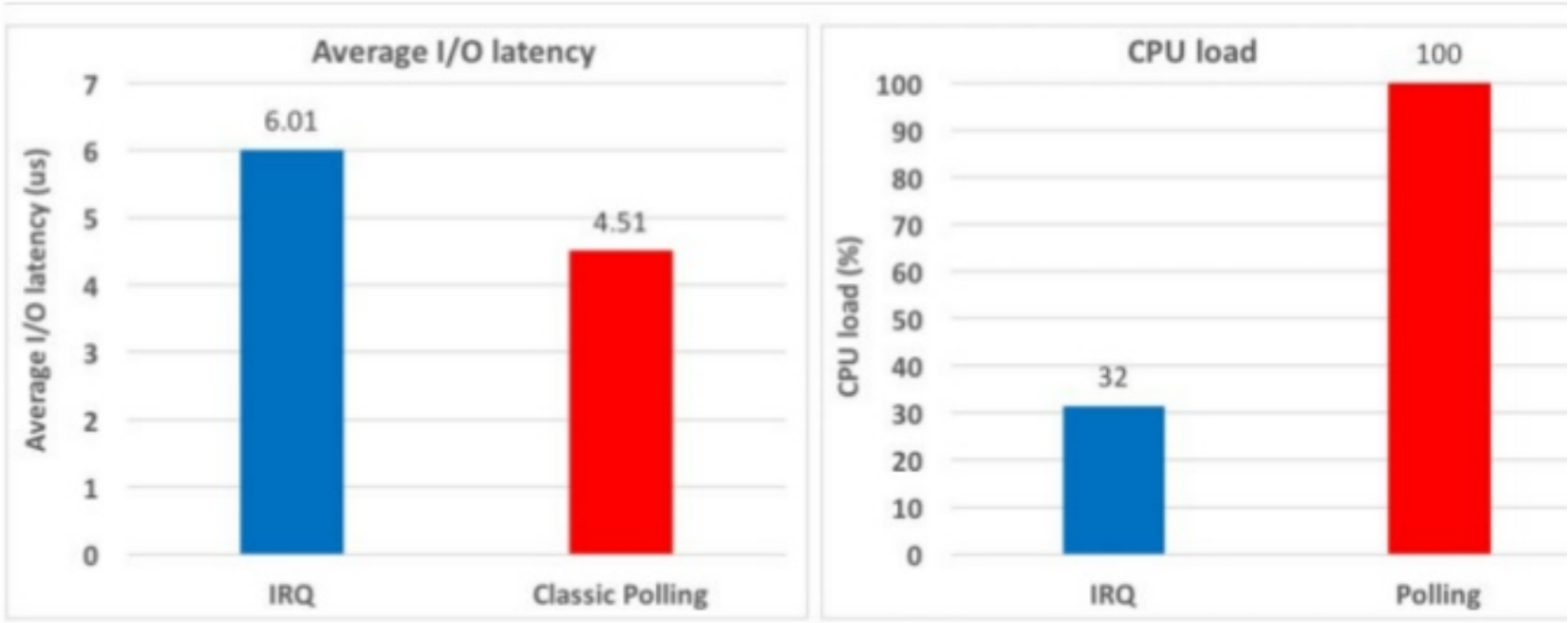
Polling based I/O completion model



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

So we should support polling!

- Add selective polling syscall interface:
  - Use `preadv2/pwritev2` with flag `IOCB_HIGHPRI`
  - Saves roughly 25% of added latency

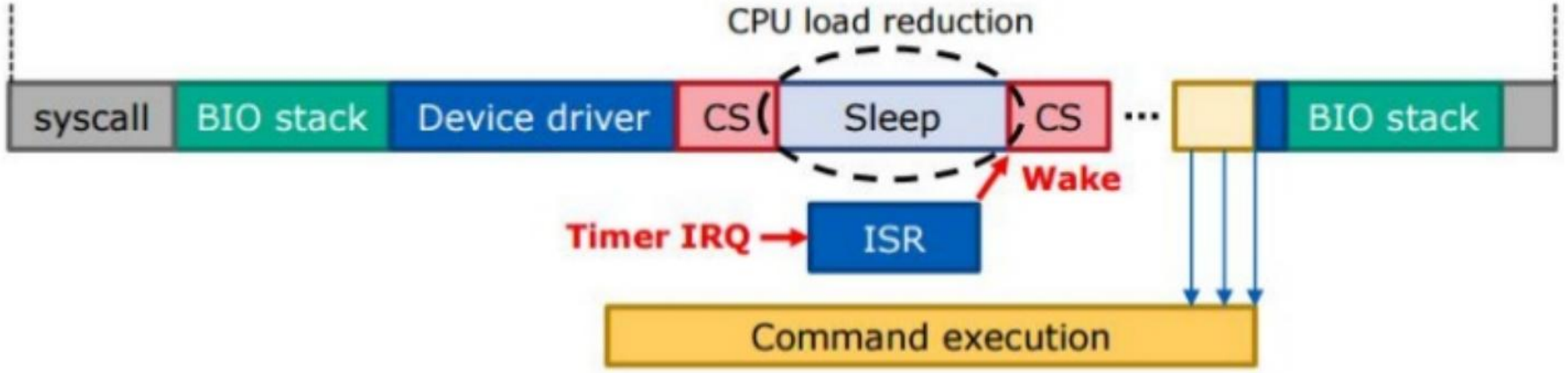


[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

But what about CPU% - can we go hybrid?

- Yes!
  - We have all the statistics framework in place, let's use it for hybrid polling!
  - Wake up poller after ½ of the mean latency.

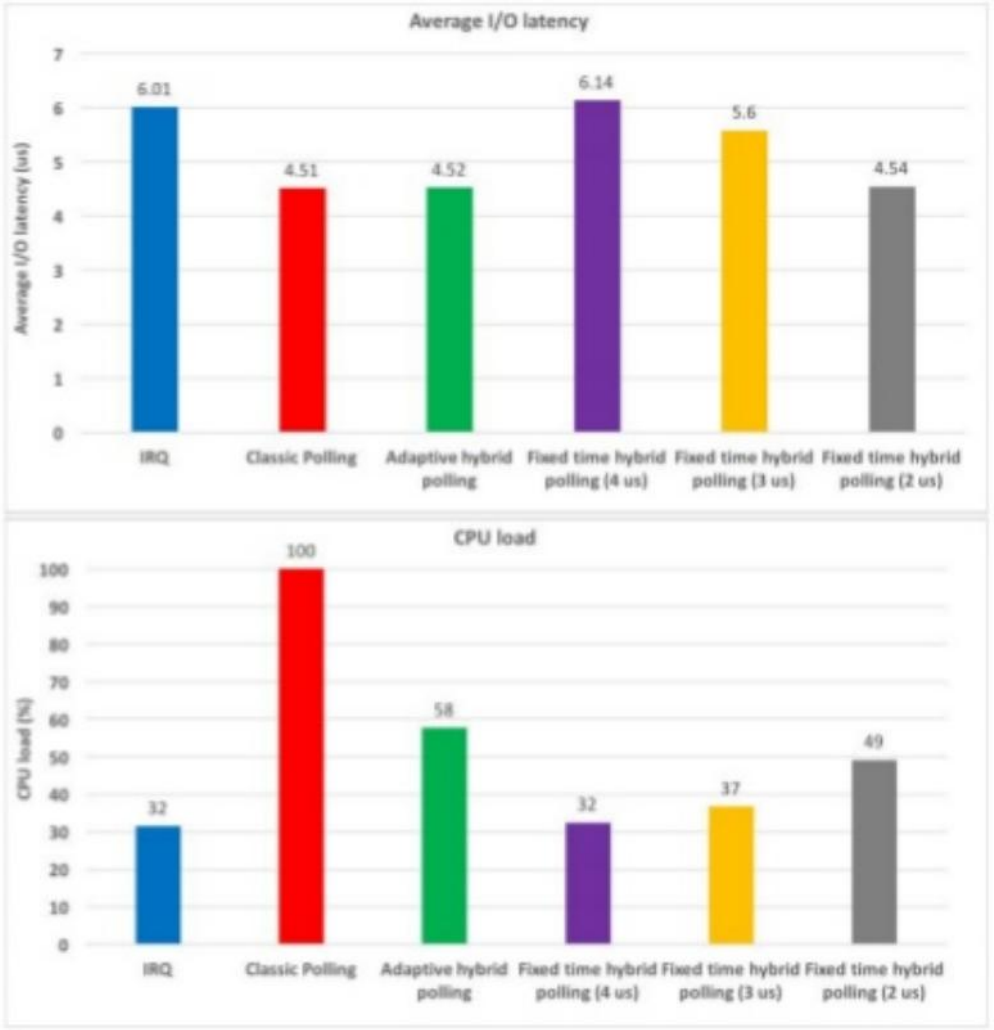
Hybrid Polling



[The Linux Block Layer. Built for Fast Storage](), Sagi Grimberg, June 2018

Hybrid polling - Performance

Polling has been added in v4.1.



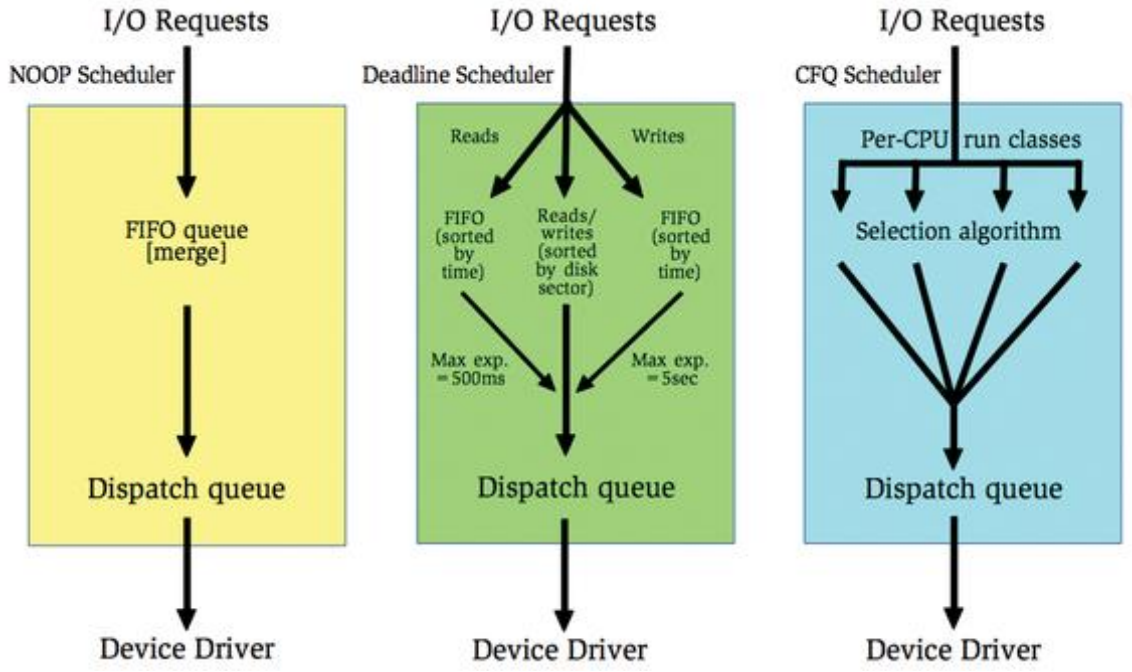
The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018

Goals of I/O schedulers

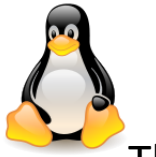
I/O schedulers can have many purposes depending on the goals; common purposes include the following:

- To **minimize time wasted** by hard disk seeks.
- To **prioritize** a certain **processes**' I/O requests.
- To give a share of the **disk bandwidth** to each running process.
- To guarantee that certain requests will be issued **before** a particular **deadline**.

(*by Wikipedia*)



[Linux Storage Stack, 2016]



I/O schedulers in Linux

The history of I/O schedulers in Linux:

- **Linux 2.4**: Linus Elevator.
- **Linux 2.6**: Deadline, Anticipatory (removed in 2.6.33), CFQ (Complete Fairness Queueing), Noop (No operation).
- **Linux 3.0**: Deadline, CFQ, Noop.
- **Linux 4.11**: MQ-Deadline, CFQ, Noop.
- **Linux 4.12**: MQ-Deadline, CFQ, BFQ (Budget Fair Queueing), Kyber, Noop.
- **Linux 5.0**: MQ-Deadline, BFQ (Budget Fair Queueing), Kyber, Noop.

I/O scheduler can be selected at boot time: **elevator**=<name of I/O scheduler> (eg. CFQ)

The administrator can replace the I/O scheduler while the kernel is running.

```
$>  cat  /sys/block/hdc/queue/scheduler
noop deadline [cfq]
$>  echo noop > /sys/block/hdc/queue/scheduler
$> ls /sys/block/hdc/queue/iosched/
back_seek_max  fifo_expire_sync  quantum  slice_idle  back_seek_penalty  group_idle slice_async  slice_sync
fifo_expire_async  low_latency  slice_async_rq  target_latency
$> cat /proc/version
$> Linux version 4.9.122-1
```

(*on **duch but 4 years ago***)

(*on **students***)

```
$> cat /sys/block/vdc/queue/scheduler
[none] mq-deadline
$ cat /proc/version
Linux version 6.1.0-21-amd64
```

(*on **duch***)

```
$> cat /sys/block/vdc/queue/scheduler
[mq-deadline] none
```

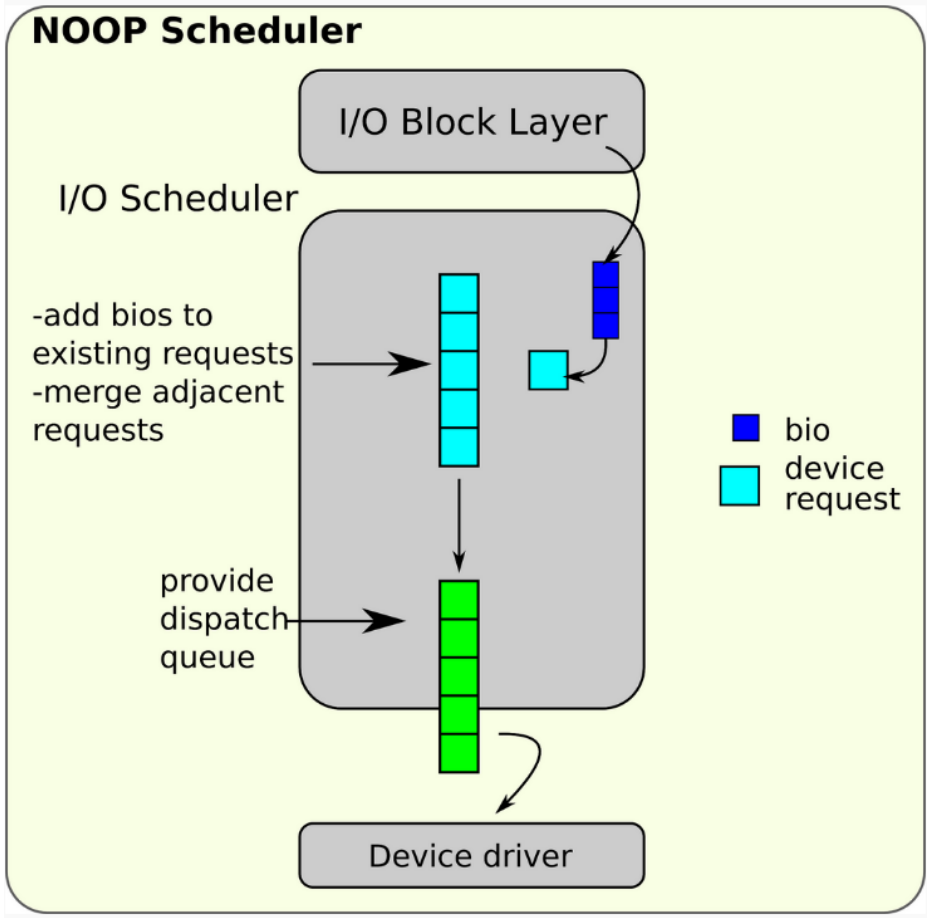
Simple (non multiqueue) I/O schedulers

Noop

Inserts all incoming I/O requests into a simple FIFO queue and implements request merging. It doesn't sort the requests

Tests by RedHat: *The noop scheduler is suitable for devices where there are no performance penalties for seeks. Examples of such devices are ones that use flash memory. Noop can also be suitable on some system setups where I/O performance is optimized at the block device level, with either an intelligent host bus adapter, or a controller attached externally.*

In Wikipedia:
https://en.wikipedia.org/wiki/Noop_scheduler



https://www.thomas-krenn.com/de/wiki/Linux_I/O_Scheduler

Simple (non multiqueue) I/O schedulers

Deadline

Tries to provide **fairness** (avoid starvation) while maximizing the global throughput.

Each request is given an **expiration time**, the deadline:

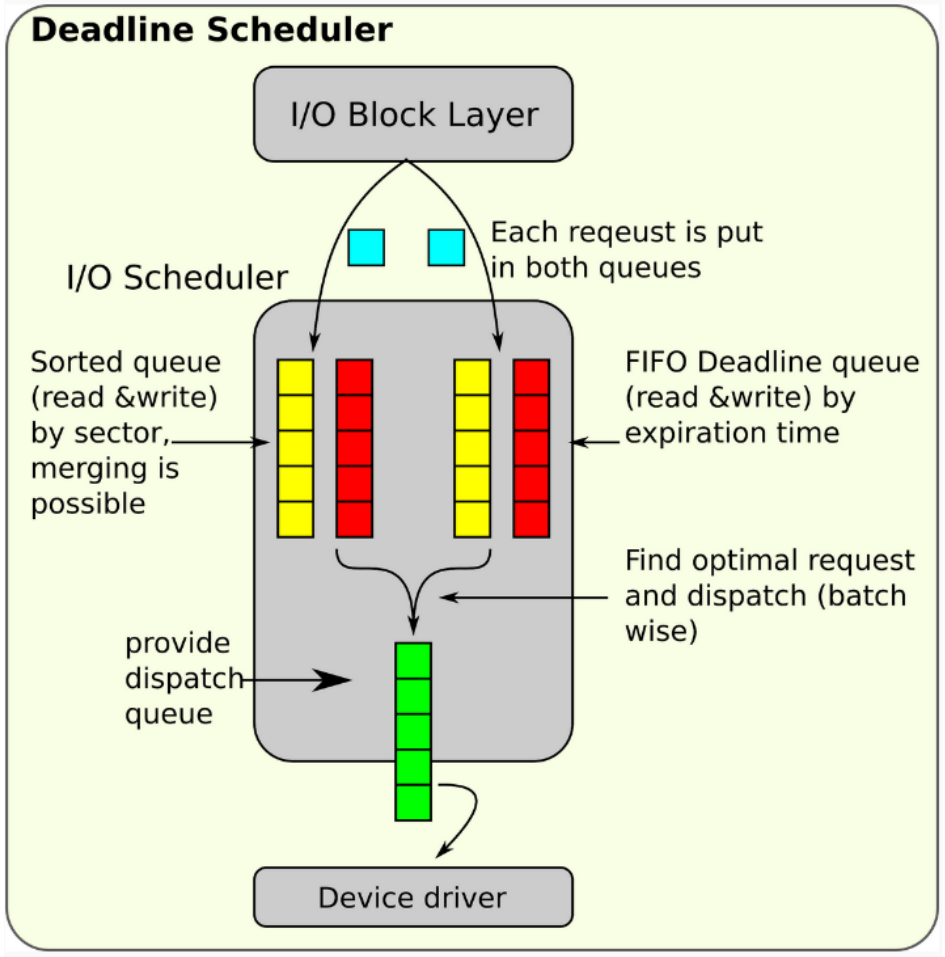
Reads = now + 0,5 sec

Writes = now + 5 sec

The algorithm **reduces latency** for **read** requests, but does so at the **expense** of **global bandwidth**.

Tests by RedHat: *The deadline scheduler aims to keep latency low, which is ideal for real-time workloads.*

In Wikipedia:
https://en.wikipedia.org/wiki/Deadline_scheduler



https://www.thomas-krenn.com/de/wiki/Linux_I/O_Scheduler

Simple (non multiqueue) I/O schedulers

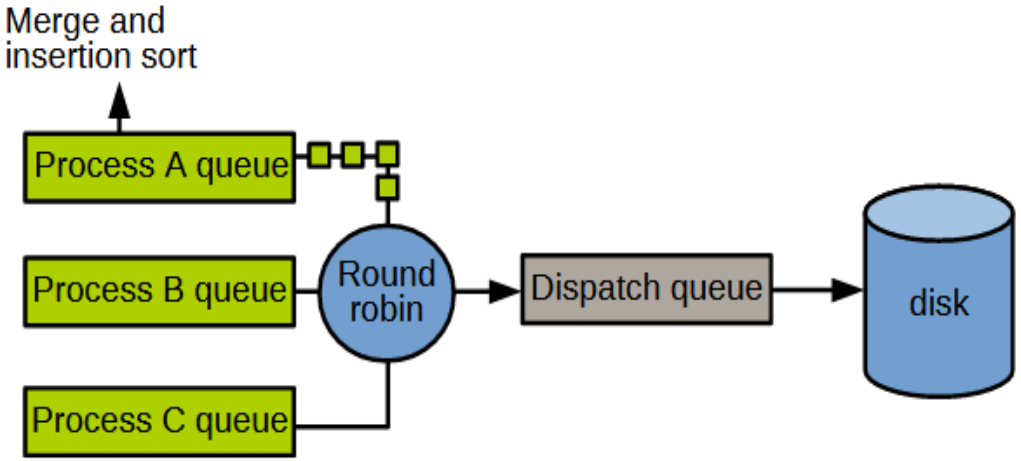
CFQ (Complete Fair Queueing)

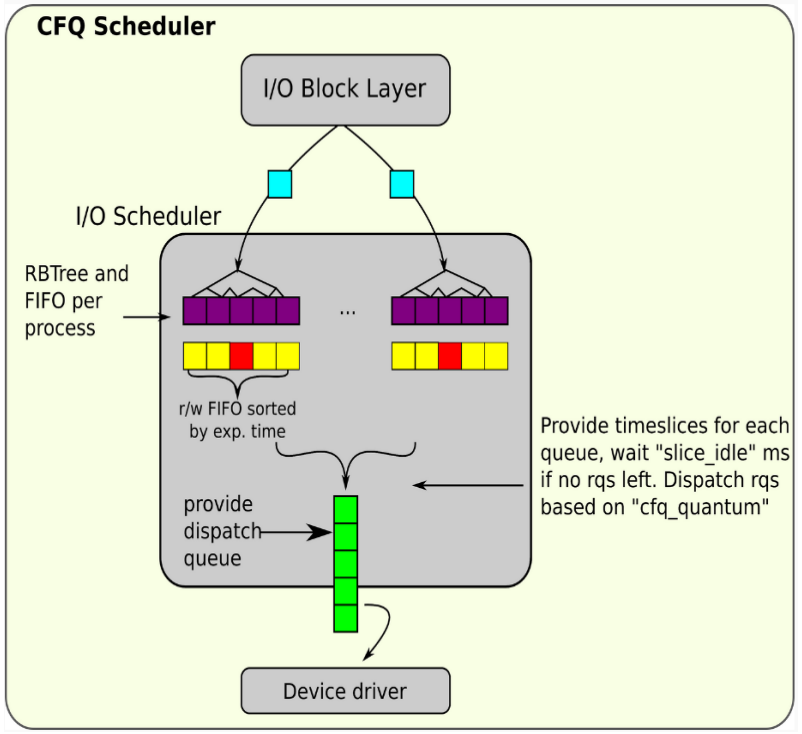
In Wikipedia: https://en.wikipedia.org/wiki/CFQ

The main goal is to ensure a fair distribution of I/O bandwidth between processes.

Per-process request queues (64 by default), which are served in round-robin order. A **hash function** is called that converts the process thread group ID (usually **PID**) into an **index** in the queue

Tests by RedHat: *CFQ is well suited for most medium to large multi-processor systems and for systems which require balanced I/O performance over multiple Logical Unit Numbers and I/O controllers.*





Source: https://computing.ece.vt.edu/~changwoo/ECE-LKP-2019F/l/lec22-block-layer

Multiqueue I/O schedulers

The **multiqueue block layer subsystem (blk-mq)**, introduced in 2013, was a step for the kernel to scale to the **fastest storage devices on large systems**. The implementation in current kernels was incomplete, in that it lacked an I/O scheduler designed to work with **multiqueue devices**.

High-end drives are generally **solid-state devices lacking rotational delay problems**; they are thus not as sensitive to the ordering of operations.

There is value in I/O scheduling even for the fastest devices; a scheduler can **coalesce adjacent requests**, reducing the overall operation count, and it can **prioritize some operations over others**.

First MQ conversion was for deadline scheduler (→ MQ-Deadline in **4.11**)

Two multiqueue I/O schedulers were added to the mainline in **4.12**.

In **5.0** the legacy (**non-multiqueue**) block layer code has been **removed**, now that no drivers require it. The legacy I/O schedulers (**including CFQ and deadline**) have been **removed** as well.

Multiqueue I/O schedulers

Budget Fair Queuing (BFQ) scheduler

Maintains **per-process queues** of I/O requests like CFQ, but it does away with the round-robin approach used by CFQ.

It assigns an **I/O budget** to each process, which is expressed as the **number of sectors** (instead of **amount** of **time** as in CFQ) that the process is allowed to **transfer** when it is next scheduled for access to the drive. Once a process is selected, it has **exclusive access** to the storage device until it has transferred its **budgeted number** of **sectors**.

The calculation of the budget is complicated, but, in the end, it is based on each **process's I/O weight** and observations of the process's past behavior.

The **I/O weight** functions like a **priority** value; it is set by the administrator (or by default) and is normally constant. Processes with the **same weight** should all get the same **allocation of I/O bandwidth**.

Different processes may get different budgets, but BFQ tries to preserve **fairness** overall, so a process getting a **smaller budget** now will get another **turn** at the **drive sooner** than a process that was given a **large budget**.

To figure out whose requests should be serviced, BFQ examines the assigned budgets and chooses the process whose **I/O budget** would, on an otherwise idle disk, **be exhausted first**.

Multiqueue I/O schedulers

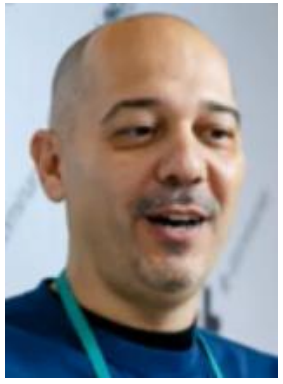
Budget Fair Queuing (BFQ) scheduler – continued

By setting the budgets properly, BFQ tries to ensure that **all processes** get a **fair amount** of the available **I/O bandwidth** within **fairly tight latency limits**.

The core idea is simple, but the real world is rather more complex; as a result, BFQ, like CFQ, has accrued a number of **heuristics aimed at improving performance in specific areas**.

- Queues for **cooperating processes** working in the same area of the disk are merged to enable more request consolidation.
- If a process doing **read** requests **empties** its queue before **exhausting** its **budget**, the device will be **idled** briefly to give that process a chance to create another request.
- Processes identified as "**soft realtime**" (those with moderate bandwidth requirements and an observed regular on/off request cycle) will get **higher priority**, as will processes that are just starting up.
- And so on.

BFQ I/O scheduler. More throughput, control and efficiency, Paolo Valente, 2019



Multiqueue I/O schedulers

Kyber I/O scheduler

BFQ is a **complex** scheduler designed to provide **good interactive** response, especially on **slower** devices. It has a relatively **high per-operation overhead**, which is justified when the I/O operations are slow and expensive.

This complexity may not make sense when **I/O operations are cheap** and **throughput** is a primary concern. When running a server workload using **solid-state** devices, it may be better to run a **simpler** scheduler that allows for **request merging** and some **simple** policies, but **mostly stays out of the way**.

Kyber is intended for **fast multiqueue devices** and **lacks much of the complexity found in BFQ**; it is **less** than **1,000 lines** of code.

I/O requests passing through Kyber are split into two primary queues, one for **synchronous requests** (reads) and one for **asynchronous requests** (writes). Reads are prioritized over writes, but not to the point that writes are starved.

The **number of operations** (both reads and writes) sent to the **dispatch queues** is **strictly limited**, keeping those queues relatively short.

Multiqueue I/O schedulers

Kyber I/O scheduler – continued

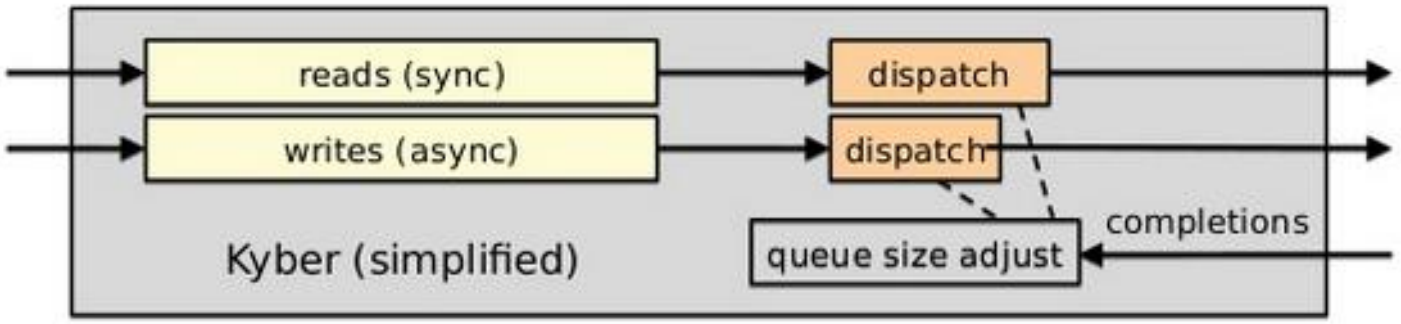
If the dispatch queues are short, the amount of time that passes while any given request waits in the queues (the **per-request latency**) will be small.

That ensures a **quick completion time** for **higher-priority requests**.

The scheduler tunes the number of requests allowed into the dispatch queues by measuring the **completion time** of each request and adjusting the limits to achieve the **desired latencies**.

Two tuning knobs are available to the system administrator for setting the **latency targets**: they default to **2 ms** for **read requests** (**read_lat_nsec**) and **10 ms** for **writes** (**write_lat_nsec**).

Tradeoffs: setting them **too low** will ensure **low latencies**, but at the **cost** of **reducing** the **opportunities** for the **merging** of requests, hurting **throughput**.



Omar Sandoval

Multiqueue I/O schedulers

Summary

Users concerned with **interactive response** and using **slower devices** will likely opt for **BFQ**.

Throughput-sensitive server loads are more likely to run with **Kyber**.

Testing I/O schedulers

- Linux 5.0 I/O Scheduler Benchmarks On Laptop & Desktop Hardware
  (Michael Larabel, February 2019)
- Improving the performance of the BFQ I/O scheduler (Paolo Valente, March 2019)
- Linux 5.6 I/O Scheduler Benchmarks: None, Kyber, BFQ, MQ-Deadline (Michael Larabel, April 2020)

The bio structure

More IOPS with BIO caching, Jonathan Corbet, **September 2021**

A **BIO** must be allocated, managed, and eventually freed for every I/O operation executed by the system. A large, busy system with fast block devices can generate **millions of I/O operations per second** (IOPS).

It turns out, that the **slab allocator is not fast enough**; it has become a bottleneck slowing down block I/O.

Jens Axboe designed a **simple cache of BIO structures**. It is built as a set of linked lists, one for each CPU in the system. When a new BIO is needed, the linked list for the current CPU is checked; if a free BIO is found there, it can be removed from the list and used without having to call into the slab allocator.

Axboe also **replaced the memset()** call with a series of statements explicitly setting each BIO field to zero. This change halves the time it takes to allocate and initialize a BIO.

With these changes in place, the block layer's performance increased by about **10%**; it can now execute over **3.5 million IOPS on each CPU core** on his test system.

Newest Linux Optimizations Can Achieve **10M IOPS Per-Core** With IO_uring, Phoronics, October 2021

Additional reading

- LWN: Driver porting: the BIO structure, Jonathan Corbet, March 2003.
- LWN: Variations on fair I/O schedulers, Goldwyn Rodrigues, December 2008.
- Linux Block IO: Introducing Multi-queue SSD Access on Multi-core Systems. M. Bjorling, J. Axobe, D. Nellans, P. Bonnet, 2013.
- LWN: The multiqueue block layer, Jonathan Corbet, June 2013.
- LWN: The BFQ I/O scheduler, Jonathan Corbet, June 2014.
- Linux kernel IO subsystem: How it works and how can I see what is it doing?, Kernel Recipes 2015, Jan Kara
- Solving the Linux storage scalability bottlenecks (slides), Kernel Recipes 2015 , Jens Axboe.
- Introduction to the Linux Block I/O Layer, Johannes Thumshin, 2016.
- LWN: The return of the BFQ I/O scheduler, Jonathan Corbet, February 2016.

Additional reading

- LWN: A block layer introduction part 1: the bio layer, Neil Brown, October 2017.
- LWN: A block layer introduction part 2: the request layer, Neil Brown, November 2017.
- LWN: Two new block I/O schedulers for 4.12, Jonathan Corbet, April 2017.
- What's new in the world of storage for Linux, Kernel Recipes 2017, J. Axboe.
- Recent Developments in The Linux IO Stack (talk, slides), M. Petersen, SDC 2017.
- Linux Optimizations for Low-Latency Block Devices (talk, slides), S. Baytes, SDC, 2017.
- LWN: I/O scheduling for single-queue devices, Jonathan Corbet, October 2018.
- The Linux Block Layer. Built for Fast Storage, Sagi Grimberg, June 2018.
- LWN: Improving the performance of the BFQ I/O scheduler, Paolo Valente, March 2019.
- Linux Multi-Queue Block IO Queueing Mechanism (blk-mq) Details, Werner Fischer, May 2020.
- Documentation/block/bfq-iosched.rst
- Documentation/block/deadline-iosched.rst